TOWARDS THE KNOWLEDGE LEVEL IN SOAR: THE ROLE OF THE ARCHITECTUR IN THE USE OF KNOWLEDGE

Technical Report AIP - 65

P.S. ROSENBLOOM, A. NEWELL, & J.E. LAIRD

University of Southern California,
Carnegie Mellon University, &
University of Michigan

August 7 1989

The Artificial Intelligence and Psychology Project

Departments of
Computer Science and Psychology
Carnegie Mellon University

Learning Research and Development Center
University of Pittsburgh

2

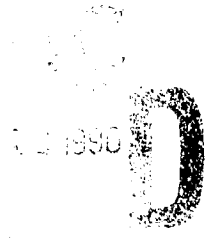
TOWARDS THE KNOWLEDGE LEVEL IN SOAR: THE ROLE OF THE ARCHITECTURE IN THE USE OF KNOWLEDGE

Technical Report AIP - 65

P.S. ROSENBLOOM, A. NEWELL, & J.E. LAIRD

University of Southern California,
Carnegie Mellon University, &
University of Michigan

August 7, 1989



This research was sponsored by the Defense Advanced Reaearch Projects Agency (DOD) under contract numbers N00039-86C-0033 (via subcontract from the Knowledge Systems Laboratory, Stanford University) and F33615-87-C-1499 (ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory), by the National Aeronautics and Space Administration under cooperative agreement numbers NCC 2-538 and NCC 2-517, and the Office of Naval Research under contract numbers N00014-86-K-0678 (Information Sciences Division) and N00014-88-K-0554 (Computer Science Division). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either exp·essed or implied, of the Defense Advanced Research Projects Agency, the National Aeronautics and Space Administration, the Office of Naval Research or the US Government.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AIP - 65	5. MONITORING ORGANIZATION REPORT NUMBER(S) Same as Performing Organization
6a. NAME OF PERFORMING ORGANIZATION Carnegie Mellon University 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Personnel and Training Research Office of Naval Research (Code 1142PT)
6c. ADDRESS (City, State, and ZIP Code) Department of Psychology Pittsburgh, Pennsylvania 15213	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, VA 22217-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Same as Monitoring Organization 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-88-K-0086
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. N/A; PROJECT NO. N/A; TASK NO. N/A; WORK UNIT ACCESSION NO N/A

11. TITLE (Include Security Classification)
Towards the knowledge level in Soar: The role of the architecture in the use of knowledge

12. PERSONAL AUTHOR(S)
Paul S. Rosenbloom, Allen Newell, and John E. Laird

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM 86Sept15 TO 91Sept14	august 7, 1989	71

16. SUPPLEMENTARY NOTATION
To appear in VanLehn, K. (Ed.), Architctures for intelligence. Hillsdale, NJ: Erlbaum.

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
procedural knowledge
episodic knowledge
declarative knowledge

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

SEE REVERSE SIDE

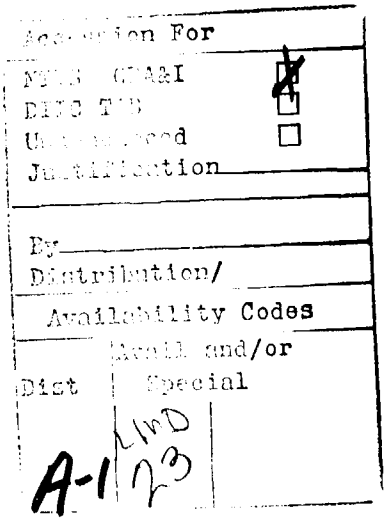
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL Susan Chipman	22b TELEPHONE (Include Area Code) (202) 696-4322 22c. OFFICE SYMBOL 1142 PT

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ABSTRACT

Soar has been described as an architecture for a system that is to be capable of general intelligence. One way to specify what this might mean is to define general intelligence as the ability to approximate an ideal knowledge level system across a sufficiently broad set of goals and knowledge. In this chapter we use this definition as the basis for evaluating the scope of this chapter, so we focus more narrowly on how the Soar architecture supports and constrains the representation, storage, retrieval, use and acquisition of three pervasive forms of knowledge: procedural, episodic, and declarative knowledge. The analysis reveals that Soar adequately supports procedual knowledge - to some extent it was designed for this - but that there are still significant questions about episodic and declarative knowledge. These questions arise primarily because of consequences of the principle source on constraint in Soar, the fact that all learning occurs via chunking. New results are also presented on the acquisition of declarative knowledge.



(A)

Towards the Knowledge Level in Soar: The Role of the Architecture in the Use of Knowledge

Paul S. Rosenbloom
Information Sciences Institute
University of Southern California

Allen Newell
Department of Computer Science
Carnegie-Mellon University

John E. Laird
Department of Electrical Engineering and Computer Science
University of Michigan

August 7, 1989

Abstract

Soar has been described as an architecture for a system that is to be capable of general intelligence. One way to specify what this might mean is to define general intelligence as the ability to approximate an ideal knowledge level system across a sufficiently broad set of goals and knowledge. In this chapter we use this definition as the basis for evaluating the degree to which Soar achieves general intelligence. A complete evaluation is beyond the scope of this chapter, so we focus more narrowly on how the Soar architecture supports and constrains the representation, storage, retrieval, use and acquisition of three pervasive forms of knowledge: procedural, episodic, and declarative knowledge. The analysis reveals that Soar adequately supports procedural knowledge – to some extent it was designed for this – but that there are still significant questions about episodic and declarative knowledge. These questions arise primarily because of consequences of the principle source of constraint in Soar, the fact that all learning occurs via chunking. New results are also presented on the acquisition of declarative knowledge. (KR)

Table of Contents

Towards the Knowledge Level in Soar: The Role of the Architect the Use of Knowledge[1]

Soar has been described as an architecture for a system that is to be capable of general intelligence (Laird, Newell, & Rosenbloom, 1987). One way to specify what this might mean is to enumerate the set of capabilities that, based on the field's cumulative experience, appear to be required for general intelligence: to be able to work on the full range of tasks, to be able to use the full range of problem-solving methods and varieties of knowledge, to be able to interact with the outside world in real time, and to learn about the world and the system's own performance. Progress can then be evaluated by determining the degree to which the architecture supports such capabilities. For Soar, such an evaluation reveals significant progress in the areas of tasks (Laird, Newell, & Rosenbloom, 1987), problem-solving methods (Laird & Newell, 1983, Laird, 1983) and learning (Steier *et al*, 1987); some progress in the area of outside interaction (Laird, Yager, Tuck, & Hucka, 1989); and an unclear situation in the area of knowledge.

---

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract numbers N00039-86C-0033 (via subcontract from the Knowledge Systems Laboratory, Stanford University) and F33615-87-C-1499 (ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory), by the National Aeronautics and Space Administration under cooperative agreement numbers NCC 2-538 and NCC 2-517, and the Office of Naval Research under contract numbers N00014-86-K-0678 (Information Sciences Division) and N00014-88-K-0554 (Computer Science Division). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Aeronautics and Space Administration, the Office of Naval Research or the US Government.

The problem with such an approach to specifying (and evaluating progress towards) general intelligence is the lack of theoretical justifications for the set of capabilities included. Without such justifications it is unclear, for example, whether some new form of learning that is developed is necessary for general intelligence, or just an interesting oddity. In addition, whole categories of critical capabilities may be unknowingly omitted. What is needed is a more fundamental definition of general intelligence from which the required capabilities can be derived (or at least justified).

One idea that shows promise towards providing such a definition is the *knowledge level* (Newell, 1981). The idea of the knowledge level is based on earlier developments in the area of computer systems levels (Bell & Newell, 1971). A computer systems level consists of a *medium* that is processed, *components* that provide primitive processing, *laws of composition* that permit components to be assembled into *systems*, and *laws of behavior* that determine how system behavior depends on the component behavior and the structure of the system. Existing levels (and their media) include the device level (electrons), the circuit level (current), the logic level (bits), the register-transfer level (bit-vectors), and the program (or symbol) level (symbols, expressions). In terms of these levels, an *architecture* is a register-transfer level system that defines a symbol level.

The knowledge level is a distinct computer systems level that lies

immediately above the symbol level. The medium processed at the knowledge level is knowledge. An agent – a system at the knowledge level – consists of a physical body that can interact with an environment, knowledge, and a set of goals. The law of behavior is the *Principle of Rationality:* "If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action." (Newell, 1981, p. 8) Once knowledge is acquired, it is available for all future goals. There are no capacity limitations on the amount of knowledge that can be available or on the agent's ability to bring it to bear in the selection of actions that achieve its goals. An essential feature of the knowledge level is that the agent's behavior is determined by the content of its knowledge, not by any aspects of its internal structure. It abstracts away from the processing and representation of the lower levels. This lack of significant internal structure implies that there are no laws of composition at the knowledge level.

The knowledge level provides a straightforward, though not uncontroversial, definition for intelligence. A system is intelligent to the degree that it approximates a knowledge-level system (Newell, 1989). Perfect intelligence requires a complete lack of internal resource limitations. However, this ideal is unreachable in physically realizable systems that are required to make decisions using bounded resources over a sufficiently wide range of goals using large bodies of knowledge. Such systems can at best only approximate a knowledge-level system, and thus

achieve some level of intelligence that is less than perfect. The ideal of perfect intelligence also does not entail the generality of that intelligence. A system's behavior is characterized both by its intelligence and by its generality. Generality for a knowledge-level system is the range of interactions that it can have with the environment, the range of goals it can have, and the range of knowledge that it can acquire and use. Intelligence is how well the system applies its knowledge to the tasks within its scope.

Assuming this knowledge-level definition of general intelligence, the key question for the architecture is how it supports the knowledge level for a sufficiently broad set of goals and knowledge. How does it approximate rationality with bounded resources? How does it support the acquisition and use of knowledge? A complete answer to the key question requires answering a number of such subquestions. In (Newell, 1989), a beginning was made at answering the first subquestion. In this chapter we provide the beginnings of an answer to the second subquestion. We examine how the Soar architecture supports and constrains the representation, storage, retrieval, use and acquisition of three pervasive forms of knowledge.

The first form of knowledge to be examined is *procedural* knowledge. Procedural knowledge is knowledge about the agent's actions. It includes knowledge about which actions can be performed, which actions should be performed when (control knowledge), and how actions are performed. The second form of knowledge to be examined is *episodic* knowledge. Episodic

knowledge is knowledge about what objects, actions, and action sequences have occurred in the agent's past. It allows answering such questions as "Did this object, action, or action sequence occur (in this context)?" and "What objects, actions, or action sequences occurred (in this context)?" The third and final form of knowledge to be examined is *declarative knowledge*. Declarative knowledge is knowledge about what is true in the world. These final two forms of knowledge have often been referred to collectively as propositional knowledge, with the term "semantic knowledge" used in place of declarative knowledge (Tulving, 1983).

The plan for this chapter is to start with a brief conventional description of the Soar architecture (Section 1), followed by its redescription in terms of the direct support it provides for knowledge (Section 2). The core of the chapter then consists of in-depth analyses of how procedural, episodic, and declarative knowledge are represented, stored, retrieved, used, and acquired in Soar (Sections 3-5). Special emphasis is placed on how the architecture supports and constrains these abilities. The chapter is concluded with a summary of key points and important directions for future work (Section 6).

1. Soar[2]

Research on Soar to date has focused on the development (and application) of an architecture for intelligence that is based on formulating

[2]This section describes Soar 4.5 (Laird *et al.* 1989), which is the basis for the analyses in this chapter

all symbolic goal-oriented behavior as search in problem spaces. The problem space determines the set of states and operators that can be used during the processing to attain a goal. The states represent situations. There is an initial state, representing the initial situation, and a set of desired states that represent the goal. An operator, when applied to a state in the problem space, yields another state in the problem space. The goal is achieved when a desired state is reached as the result of a sequence of operator applications starting from the initial state. Each goal defines a problem-solving context ("context" for short) that contains, in addition to a goal, roles for a problem space, a state, and an operator.

Problem solving for a goal is driven by decisions that result in the selection of problem spaces, states, and operators for the appropriate roles in the context. Decisions are made by the retrieval and integration of preferences – special architecturally interpretable elements that describe the acceptability, desirability, and necessity of selecting particular problem spaces, states, and operators. The context in which a preference is applicable is specified by its goal, problem-space, state, and operator attributes. When present, they specify the objects that must be already selected in the context for the preference to be valid. For example, the following is a desirability preference stating that operator *o1* is at least as good as any other operator – that is, it is ***best*** – for state *s1*, problem space *p1*, and goal *g1*.

```
(preference o1 ^role operator ^value best
    ^goal g1 ^problem-space p1 ^state s1)
```

There are two types of acceptability preferences – acceptable and reject – to rule an operator into and out of consideration for selection. A reject preference overrides an acceptable preference. There are five types of desirability preferences – worst, worse, indifferent, better, and best – to determine the relative desirability of considered objects. Worst and best are unary preferences. Worse and better are binary preferences. Indifferent can be binary or unary, in which case the object is indifferent to all other competing objects with indifferent preferences. There are two types of necessity preferences – require and prohibit – for asserting that an object must or must not be selected for a goal to be achieved. Details on the semantics of preferences can be found in (Laird, Newell, & Rosenbloom, 1987).

All long-term knowledge is stored in a recognition-based memory – a production system. Each production is a cued-retrieval unit that retrieves the contents of its actions when the pattern in its conditions is successfully matched. By sharing variables between conditions and actions, productions can retrieve information that is a function of what was matched. By having variables in actions that are not in conditions, new objects can be generated/retrieved.

Transient process state is contained in a working memory. This includes information retrieved from long-term memory, results of decisions made by the architecture, information currently perceived from the external environment, and motor commands. It should be clear that this process

state is much more than just a single state in a problem space. The process state is the entire transient state of the system, which includes as components, states in problem spaces, and in fact whole problem-solving contexts. It provides the cues for retrieving additional information from long-term memory.

Structurally, working memory consists of a set of objects and preferences about objects. Each object in working memory has a class name, a unique identifier, and a set of attributes with associated values, which may be constants or identifiers (allowing a graph structure of objects). For example, a particular box could be represented by the following object.

(box *b1* ^name box1 ^height 10 ^width 4 ^depth 2)

The class is "box", the identifier is "*b1*", the name of the box is "box1", and the box has a height of 10 a width of 4 and a depth of 2.

For each problem-solving decision, the contents of working memory is elaborated by parallel access of long-term memory to exhaustion. All productions that match the current working memory are fired in parallel, and this repeats until no productions match. This elaboration process retrieves into working memory new objects, new information about existing objects, and new preferences. When quiescence is reached – that is, when no more productions can fire – an architectural decision procedure interprets the preferences in working memory according to their fixed semantics. If the preferences uniquely specify an object to be selected for a role in a context, such as selecting the current operator for a

state, then a decision can be made, and the specified object becomes the current value of the role. The whole process, an elaboration phase followed by a decision, then repeats.

If the decision procedure is ever unable to make a selection – because the preferences in working memory are either incomplete or inconsistent – an *impasse* occurs in problem solving because the system does not know how to proceed. When an impasse occurs, a subgoal with an associated problem-solving context is automatically generated for the task of resolving the impasse. The impasses, and thus their subgoals, vary from problems of selection (of problem spaces, states, and operators) to problems of generation (e.g., operator application). Given a subgoal, Soar can bring its full problem-solving capability and knowledge to bear on resolving the impasse that caused the subgoal. For example, if an operator-tie impasse occurs because multiple operators are competing for selection with insufficiently distinguishing preferences, then a subgoal is created in which Soar can (among other things) execute operators to evaluate the competing alternatives. Productions can then create preferences based on these evaluations, allowing the decision to be made.

When impasses occur within impasses – if, for example, there is insufficient knowledge about how to evaluate a competing alternative – then subgoals occur within subgoals, and a goal hierarchy results (which therefore defines a hierarchy of contexts). The top problem space consists of task operators: such as, to recognize an item. The subgoals are

generated as the result of impasses in problem solving. A subgoal terminates when the impasse is resolved.

Soar learns by acquiring new productions that summarize the processing that leads to the results of subgoals, a process called *chunking*. The actions of the new productions are based on the results of the subgoal. The conditions are based on those working memory elements in supergoals that were relevant to the determination of the results. Relevance is determined by using the traces of the productions that fired during the subgoal. Starting from the production trace that generated the subgoal's result, those production traces that generated the working-memory elements in the conditions of the trace are found, and then the traces that generated their condition elements are found, and so on until elements are reached that are in supergoals. Productions that only generate desirability preferences do not participate in this backtracing process – desirability preferences only affect the efficiency with which a goal is achieved, and not the correctness of the goal's results.

Soar's perceptual-motor behavior is driven by a set of asynchronous modules, and mediated through the state in the top context. Each perceptual and motor modality (module) has its own state attribute to which perceptual information is added and/or motor commands are taken. New sensory information arrives in working memory whenever it is available, and motor commands are sent to the appropriate motor modules as soon as they are added to working memory. Sensory

information can be retained by explicitly attaching it to other existing structures. Otherwise, it will be displaced when new information arrives.

Figure 1-1 summarizes the major functional and structural components of the Soar architecture – its memories, basic computational cycle, learning, and interfaces.

- Purpose of Research: Architecture for general intelligence.
- Organizing Framework: Goals and problem spaces
- Long-term Memory: Recognition-based productions.
- Short-term Memory: Objects and attributes.
- Basic Computation Cycle: Elaboration (access LTM until quiescence) and decision.
- Decisions: Preference-based for problem spaces, states, and operators.
- Subgoal Creation: Impasses in decision scheme.
- Learning: Chunking – summarize processing of subgoal as a production.
- Interface to External Environment: Asynchronous through top-state.

Figure 1-1: Summary of Soar.

2. Architectural Support for Knowledge

The conventional description of Soar provided in the previous section does not always make clear the ways in which the architecture directly supports knowledge. That is the task for this section – to make explicit the ways the architecture directly supports knowledge in general, and procedural, episodic, and declarative knowledge in particular. The question of indirect architectural support is left to the later sections, which examine each of these three types of knowledge in detail.

General support is provided by productions, the elaboration phase, impasses, subgoals, problem space search, working memory, and chunking. Productions provide for the explicit storage of knowledge. The knowledge is stored in the actions of productions, while the conditions act as access paths to the knowledge. The process of retrieving knowledge by the matching and firing of a production comprises a search of the system's explicitly stored long-term knowledge. It is thus termed *knowledge search* (or *k-search*). Knowledge retrievable by k-search – i.e., by the firing of a production – is termed *k-retrievable knowledge*. K-search is efficient, but relatively limited in its capabilities.

Knowledge that is not retrievable by the firing of a single production may still be retrievable by the firing of multiple productions in a single elaboration phase. This happens when information retrieved early in an elaboration phase provides the cues that allow the desired information to be retrieved by a later production firing. It also happens when the desired

information is distributed among the actions of multiple productions, which retrieve it by firing jointly within the same elaboration phase. This is termed *k*-search*, and knowledge retrievable through elaboration is termed *k*-retrievable knowledge*. k*-search is exhaustive but efficient, allowing the system to use a significant body of knowledge in its decisions even under relatively stringent time constraints.

The creation of impasses provides a means for determining when the k*-retrievable knowledge is an inadequate basis for making a decision. The decision procedure can detect incompleteness and inconsistency in the set of k*-retrievable preferences, but cannot directly detect incorrect or sub-optimal knowledge.

Subgoals provide contexts in which knowledge that is not k*-retrievable can be retrieved by *problem-space search* (or *ps-search*). Knowledge that is retrievable by ps-search is termed *ps-retrievable knowledge*. Because problem-space search (ps-search) is always eventually grounded in production firings (k*-search), there is a fairly direct relationship between ps-retrievable knowledge and k*-retrievable knowledge.[3] Knowledge that is ps-retrievable in the current context is constructed from pieces of knowledge which are independently k*-retrievable in other contexts, but not jointly k*-retrievable in the current context. Ps-search allows for the

[3]Here, and in the remainder of this chapter the terms k*-search and k*-retrievable knowledge will be assumed to subsume the terms k-search and k-retrievable knowledge, respectively, except where the distinction is particularly crucial.

consideration of alternatives and the deliberate construction of information. whereas k*-search provides for only the monotonic accumulation of knowledge. Problem-space search is selective and slow. but can with sufficient resources retrieve any knowledge in the system's knowledge level.

Working memory provides a locus where retrieved knowledge can be examined and used. It also provides a locus where new knowledge can reside temporarily before it is stored into long-term memory by chunking. Chunking provides a means of creating new productions. thus directly augmenting the system's store of k*-retrievable knowledge. and indirectly augmenting its store of ps-retrievable knowledge. Chunking is the mechanism for converting ps-search to k*-search.

Procedural knowledge is specifically supported by the architecture in four ways. First, production execution is a primitive form of controlled action. Executing a production performs a form of retrieval in which the retrieved information is adapted to the current situation before being retrieved. The nature of the adaptation is determined by the production's variables. Variables that are shared between conditions and actions result in the retrieved information being instantiated to be about existing objects. Variables that exist only in actions result in the creation of new objects. Control is exerted on production execution by the match. Production conditions specify situations that must hold in working memory in order for the retrieval actions to be executed (Newell.

Rosenbloom & Laird, 1989). Unlike traditional production systems, there is no additional conflict resolution process that participates in the control of production execution.

Second, the selection of an object for a context slot is also a primitive form of controlled action. Selections are actions performed by the architecture that change the focus of problem solving in working memory. For example, the selection of a new operator changes what the system is attempting to accomplish. Preferences represent architecturally interpretable control information for the selection process.

Third, the concept of a problem-solving operator is partially supported by the architecture. The architecture provides an operator role in contexts and the decision procedure that enables the selection of operators for operator roles. It also provides for the generation of impasses when there is insufficient knowledge about how to select or execute an operator. An important form of support not provided is an architecturally interpretable operator language. Instead, operator execution always eventually grounds out in memory retrieval (and motor behavior). How this happens may be quite complicated, involving numerous subgoals, or the interpretation by productions – that is, by further memory retrieval – of an arbitrary operator language.

Fourth, the architecture provides motor commands that perform primitive actions in the external environment. There is not yet a complete

and standard set of motor commands in Soar. Instead, what exists is a text-output module, providing basic text-output commands, and a flexible mechanism for adding new modules to, for example, control robot arms (Laird, Yager, Tuck, & Hucka, 1989) and mobile robots. Selection of motor commands is not provided directly by the architecture. It is under the control of the knowledge (productions) which retrieve the motor commands into working memory, usually under the aegis of operator execution.

Episodic knowledge is specifically supported by the chunking and execution of new productions. Chunking acquires new productions based on problem-solving episodes. The actions of a chunk correspond to information that was generated as the result of an episode. When the chunk executes it retrieves information that is similar to that generated during the episode – though, as mentioned above, the retrieved information is generally adapted to the current situation rather than being a verbatim record of the earlier episode's results. The conditions of the chunk ensure that the adapted results are only retrieved in similar situations. Not provided by the architecture is a mechanism that creates verbatim records of the system's experiences for later examination.

Declarative knowledge is specifically supported by the working and production memories. Working memory is a transient memory of objects, with associated attributes and values. These objects are declarative in that they are examinable (by productions), but they need not have a fixed

semantics. Production memory provides for long-term storage of declarative structures – in the actions of productions – which can be retrieved (and adapted) by production execution.

3. Procedural Knowledge

As mentioned in the introduction, procedural knowledge is knowledge about the agent's actions, which includes knowledge about which actions can be performed, which actions should be performed when (control knowledge), and how actions are performed. Procedural knowledge is already one of the most well developed and understood parts of Soar. Soar was, after all, originally developed as a general problem-solving architecture. Thus this section primarily serves as a review, but it also serves to develop a number of the basic concepts used in the subsequent sections on episodic and declarative knowledge. The discussion is divided into subsections covering the three subdomains mentioned above: performable actions, action control, and action performance. For each of these subdomains, we discuss how the knowledge is represented, stored, retrieved, used, and acquired. This same suborganization will be followed in later sections on episodic and declarative knowledge.

3.1. Performable Actions

Performable actions are represented as operators, along with acceptable preferences that can cause the operators to be considered in some set of situations. Each operator is represented in working memory as an object – a declarative structure – rather than a production. For example, an

operator in the Eight Puzzle that slides a tile from one cell on the board to an adjacent one could be represented as (operator *o1* ^name slide). When augmented with parameters specifying the source and destination cells for the tile, the operator can be represented as (operator *o1* ^name slide ^source *c1* ^destination *c2*), where the symbols *c1* and *c2* are the identifiers of the two cells.

The declarative structure for operators, and their acceptable preferences, are stored in the actions of productions. The entire object can be stored in the actions of a single production (k-retrievable); it can be distributed across the actions of a group of productions that all fire within a single decision cycle (k*-retrievable); or it can be distributed across multiple productions that fire in a subgoal that constructs the operator, bit by bit (ps-retrievable).

Problem spaces are a major source of context for operator retrieval. The production in which the above Eight Puzzle operator is stored will have a condition which tests that the Eight Puzzle problem space is the one currently selected in a context before retrieving the operator for the context. It is also often useful to utilize the operator's preconditions as a source of retrieval context. If this is done, then the operator is only retrieved in situations for which it is applicable. An alternative is to retrieve the operator according to means-ends analysis, that is, when the operator will reduce the difference between the current state and the desired state. With means-ends analysis, an operator may be retrieved

even when its preconditions are not satisfied by the current state.

Operators by themselves do nothing. The architecture does not understand the language(s) in which operators are written, and therefore does not know how, based just on the operators themselves, to either select among them or to perform them. The best the architecture can do without additional knowledge is to perform various default actions based on its understanding of their acceptable preferences. It can select an operator if it is the only candidate available, and generate an impasse if there is more than one operator, or if the selected one cannot be executed.

Operators, and their acceptable preferences, are cues for retrieving a variety of additional knowledge. The operator structure can trigger knowledge about how to select and perform operators (Sections 3.2 and 3.3). The acceptable preferences can trigger knowledge in both prospective and retrospective fashions. Prospectively, acceptable preferences for operators determine what operators are being considered for the next selection. Retrospectively, acceptable preferences for operators act as episodic knowledge about what operators were considered for what states (Section 4).

Operators, and their acceptable preferences, are acquired by the chunking of problem-solving episodes that generate them as results. Chunking does not by itself generate new operators, but it can convert ps-retrievable operators into k*-retrievable ones, as well as store away in

production memory new operators that are generated. The conversion of operator knowledge from ps-retrievable to k*-retrievable is the obvious caching effect produced by chunking. Storage of newly generated operators factors into two cases. If the new operators are generated internally, then they must have already been ps-retrievable – that is, retrievable by problem space search – thus reducing this case to the previous caching situation. If the new operators are based on external information, chunking can turn unretrievable operators into k*-retrievable operators. This is a more subtle consequence of chunking that is worth looking at in some detail.

Yost & Newell (1988) demonstrated how new operators could be acquired from external information, in the context of a system called TAQ (Yost, 1987) that acquires new tasks (i.e., problem spaces) from external descriptions. In more recent work, this approach has been extended to take simple English instructions for a range of immediate-reasoning tasks, such as categorical syllogisms and sentence verification (Lewis, Newell, & Polk, 1989). Figure 3-1 shows the two basic steps. The first step in task acquisition is to comprehend an externally provided description of the task to be acquired. This description can conceptually take a variety of forms – versions of TAQ have accepted descriptions in simple English sentences and in a formal problem space notation. The outcome of the comprehension process is the presence in working memory of a declarative description of a problem space for the task. The second step is to solve

the problem using the declarative description interpretively. That is, at each point in task performance, if the next required activity – such as the generation of an operator – is not directly performable by k*-search, then a subgoal occurs. Within the subgoal, the declarative task description is examined and interpreted by a set of pre-existing problem spaces that search through the declarative task description for information about what to do in the current situation.

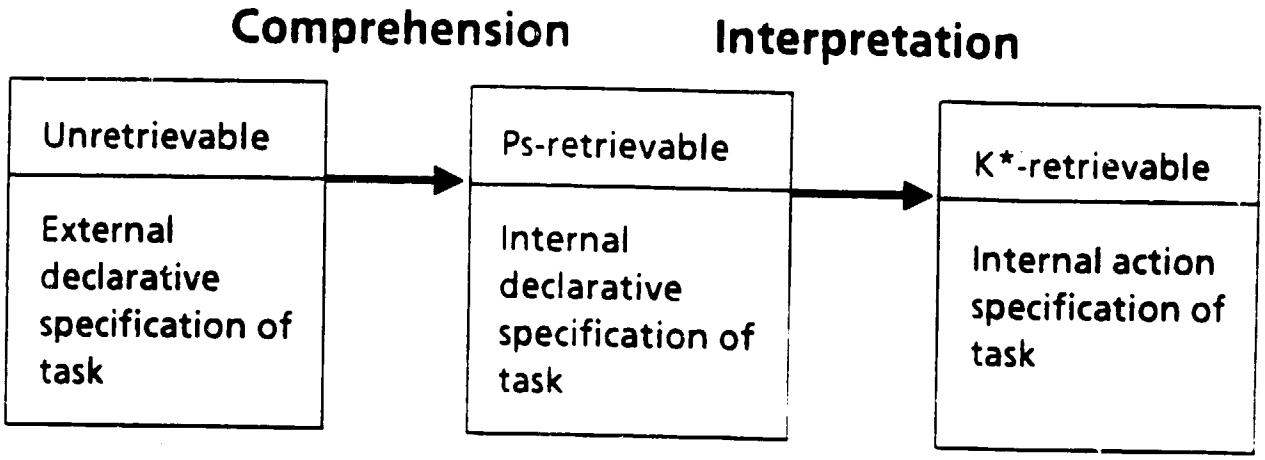


Figure 3-1: The two stages of acquiring operator knowledge.

The chunks acquired for these interpretation subgoals directly implement the required activity. Chunking the comprehension process converts unretrievable operators into ps-retrievable operators – using memorization techniques described in Sections 4 and 5 – and chunking of the interpretation process makes the ps-retrievable knowledge k*-retrievable.

3.2. Action Control

Control knowledge – that is, knowledge about how to select among performable actions – is represented to the architecture by operator preferences. The three types of preferences described in Section 1 – acceptability, desirability, and necessity – are used to represent three qualitatively different types of control knowledge. Acceptability preferences represent knowledge about whether an operator is to be considered for execution. Unconsidered operators have no effect on the decision procedure: they cannot be selected, nor can they cause an impasse to occur. Desirability preferences represent heuristic information that can be brought to bear in determining what operator is likely to lead towards goal satisfaction. Necessity preferences represent constraints derived from the goal. They can be used to guarantee that certain conditions are always (or never true) during the search, thus eliminating the need to explicitly test them at the end. In the extreme, necessity preferences can be used to explicitly represent the entire sequence of steps in a procedure that achieves some goal, eliminating the need for an explicit goal test at the end.

Preferences are stored in the actions of productions. In any particular situation an arbitrary preference can be k-retrievable, k*-retrievable, ps-retrievable, or unretrievable. The primary context for preference retrieval is the object being considered and the objects already selected as part of the problem solving context. For example, the retrieval context for

operator preferences in the Eight Puzzle includes the operator being considered (to slide a tile from cell c1 into cell c2), the goal to be achieved (a solution to the Eight Puzzle), the problem space (Eight-Puzzle), and the state to which the operator is to be applied. For binary preferences, such as better worse and indifferent preferences, the retrieval context includes muitiple contending objects.

Preferences are used both by the architecture – the decision procedure – and by other knowledge. The architecture uses preferences to determine what selection to make, or what type of subgoal to generate if no selection can be made. As mentioned in Section 3.1, preferences can act as cues about what the decision procedure is going to do, and to reconstruct what it did in the past.

Preferences are acquired by the chunking of problem-solving episodes that generate preferences as their results. Most of our experience in acquiring preferences involves the acquisition of desirability preferences and that is all that will be discussed here, though the acquisition of acceptable preferences is covered under the discussion of operator acquisition in Section 3.1.

As with knowledge about performable actions, chunking can turn ps-retrievable preferences into k*-retrievable preferences, and convert unretrievable preferences into ps-retrievable and k*-retrievable preferences. The most common way to turn ps-retrievable preferences

into k*-retrievable ones involves a look-ahead search. This process has been described in detail elsewhere (Laird, Newell, & Rosenbloom, 1987), but the essence is to use Soar's basic search capability along with knowledge – about how to evaluate states, how to back up evaluations to earlier operators and states, and how to generate preferences from evaluations – to generate, and thus learn via chunking, preferences about operators that have tied for selection.

As demonstrated in (Golding, Rosenbloom, & Laird, 1987), it is possible to use external advice to assist in the process of converting ps-retrievable knowledge into k*-retrievable knowledge. If advice is given about what alternatives are good (or bad, for that matter), the advice can be turned into preferences which guide the look-ahead search. This can reduce the amount of search required without changing the chunks that are learned for the search. Externally provided knowledge can also be used to shift a piece of control knowledge from unretrievable to k*-retrievable using the techniques described in Section 3.1 (Yost & Newell, 1988).

3.3. Action Performance

As mentioned previously, Soar does not have a single, architecturally interpretable language for action performance. Instead, there are several distinct ways of representing action performance. One way to represent action performance, at least for external actions, is as some combination of motor commands. Retrieval of motor commands into working memory causes the associated motor systems to behave in appropriate fashions.

There is a fixed language of motor commands, as determined by the available motor systems.

A second way to represent action performance is as a subprocedure that performs the action when executed. In Soar terms, the subprocedure is the processing in a subgoal that arises when the results of performing the action are not k*-retrievable. Within the subprocedure, the types of procedural knowledge described in this section would be applied recursively.

A third way to represent action performance is as the state that results from applying the operator representing the action. As with operators, the entire state can be stored in the actions of a single production (k-retrievable), it can be distributed across the actions of a group of productions that all fire within a single decision cycle (k*-retrievable), or it can be distributed across multiple productions that fire under different circumstances (ps-retrievable). The ps-retrievable case corresponds to the representation of action performance as subprocedures that is described above. Such a procedural representation – that is, representation as a subprocedure, rather than representation of a procedure – can actually be used for any piece of knowledge, whether the knowledge is itself about procedures or not.

The primary context for the retrieval of a result state is the conjunction of relevant features of the previous state and the operator. The result

state is used as the basis for further problem solving, by serving as part of the retrieval context for goal testing, result generation, state and operator evaluation, and operator generation, selection, and application. The result state's acceptable preference is used by the decision procedure to select the state as the current state. As mentioned in the previous subsections, other knowledge may also use the preference prospectively to determine what state is going to be selected, and retrospectively to determine what state was selected, and what operator and state preceded it.

Acquisition of result states occurs by the chunking of problem-solving episodes that generate such states. One of the most common ways to acquire a k*-retrievable result state is to chunk over the process of executing a procedure that represents an action. As with the acquisition of knowledge about control, external advice can be utilized to speed up the process of acquiring knowledge about action performance. In one version, demonstrated for subtraction, Tic-Tac-Toe, and simple block manipulation, the system starts out with a set of primitive operators that are sufficient to implement the individual tests and modifications made by any operator. Advice is then used to determine which elements the action should test and generate for the specific operator being acquired. Given the primitive operators, this approach allows arbitrary operators to be acquired from advice.

Another way to acquire knowledge about action performance is to chunk over the process of interpreting an externally provided description of the

action, as in (Yost & Newell, 1988). The process proceeds much as did the corresponding one in Section 3.1, where in this case, one or more chunks are learned that can retrieve the result state in the future.

3.4. Summary

Procedural knowledge appears to be adequately supported by the current architecture. This should not be too surprising as it was originally designed for this: or at least for representing problem-solving knowledge. One aspect that might come as a surprise is that productions, though they are a primitive form of action, are not the model for action – operators are. Another possibly surprising aspect is that there is no single architecturally interpreted operator language. The fixed operator language common to most systems is replaced by the ability to perform operators by memory retrieval – either k*-retrieval or ps-retrieval – in conjunction with motor commands. The flexibility of this approach allows performance knowledge to be represented either directly in action form or as declarative structures that are interpreted. In fact, with the aid of software interpreters it should be possible to construct arbitrary operator languages. One example of such an approach is the language and interpreter used in the task acquisition work.

Learning has an important place in the use of procedural knowledge. By converting ps-retrievable knowledge into k*-retrievable knowledge, it can improve the system's ability to retrieve relevant knowledge under real-time constraints, and thus improve the system's approximation to the

principle of rationality. It can convert interpreted behavior into direct action. It can also acquire new knowledge from the outside world, allowing the system to expand the tasks it can work on and the knowledge that it can use on those tasks.

4. Episodic Knowledge

Episodic knowledge is knowledge about what has occurred. In general, the individual elements of episodic knowledge can be viewed as instances of a binary predicate, Occurred(x, y), where x is an object, action, or sequence of actions that has occurred – for simplicity we will refer to all such members of the class of things that can occur as *events* – and y is a context in which the event occurred. Two loose but illustrative examples are Occurred("gaf", "List 1 of Experiment 2"), which denotes that a particular object (the nonsense trigram "gaf") occurred in a particular context (during the first list of experiment 2), and Occurred("pull-knob then turn-knob", "setting time on watch"), which denotes that a particular sequence of actions (pulling out of the watch's knob followed by turning of it) occurred in a particular context (the setting of the watch).

There are two notable features about the role of episodic knowledge in Soar. First, episodic knowledge can be represented at many different levels of explicitness. Second, although the representation, storage, retrieval, and use of episodic knowledge is rather straightforward, the acquisition of some forms of episodic knowledge is quite challenging. It leads us to posit the existence of comparatively complex strategies for

acquiring these forms of episodic knowledge.

4.1. Representation

One way to represent episodic knowledge is completely as declarative structures: that is, as objects with attributes and values. In such a representation, the above two examples might appear as follows.

(occurred *e1* 'event *o1* 'context *c1*)
(object *o1* 'name gaf)
(context *c1* 'experiment 2 'list 1)

(occurred *e2* 'event *s1* 'context *c2*)
(sequence *s1* 'action1 *a1* 'action2 *a2*)
(action *a1* 'name pull-knob)
(action *a2* 'name turn-knob)
(context *c2* 'name setting-time-on-watch)

However, not all of this knowledge need be represented directly as declarative structures. The alternative is to omit some of the components from the explicit representation, and assume them implicitly by default. The key to making this work is an understanding of the episodic nature of chunking: that is, that chunks are acquired as the result of problem-solving episodes, and execute in contexts that are similar to the ones in which they were learned. Spinning out the consequences of this understanding leads to a sequence of ways of omitting and modifying components of the representation.

The first component that can be omitted is the context. The conditions of a chunk represent both the context in which the information was learned and the contexts in which it should be retrieved. Therefore, when

information is retrieved from long-term memory it can be assumed that it was learned in a situation that was similar to the retrieval situation. Eliminating the explicit context from the example above leaves the following explicit structures.

(occurred *e1* ^event *o1*)
(object *o1* ^name gaf)

(occurred *e2* ^event *s1*)
(sequence *s1* ^action1 *a1* ^action2 *a2*)
(action *a1* ^name pull-knob)
(action *a2* ^name turn-knob)

The next component that can be omitted is the occurred predicate, which, now that the context is removed, just states that its event occurred. If it is assumed that all knowledge that is retrieved from long-term memory got there via chunking – even if the system starts out with a number of productions, after sufficient time nearly all of its productions should have been acquired by chunking – then it can be assumed that any structures retrieved from long-term memory must have been seen before. Therefore, the explicit predicate can be eliminated in favor of the assumption that anything that is retrieved has occurred. Eliminating the occurred predicate from the example leaves the following explicit structures.

(object *o1* ^name gaf)

(sequence *s1* ^action1 *a1* ^action2 *a2*)
(action *a1* ^name pull-knob)
(action *a2* ^name turn-knob)

The episodic knowledge about the gaf event is now represented quite simply as a typical Soar object – all structures that were introduced solely for their role in representing episodic knowledge have been deleted. This is thus its final form. However, for the watch event, two additional steps are needed to convert its action sequence into its final form. The first step involves a change in nomenclature to replace actions with operators. Operators are intended to represent actions that can be performed rather than actions that were performed, but the assumptions made so far imply that if an operator is retrieved in a context then it must have been learned in a similar context. Operators can therefore stand in for actions that have occurred. Making this change eliminates the need for creating new structures to explicitly represent actions that have occurred, using the existing operator structures instead.

(sequence *s1* 'operator1 *a1* 'operator2 *a2*)
(operator *a1* 'name pull-knob)
(operator *a2* 'name turn-knob)

The second, and final, step is to replace the explicit representation of operator sequences with preferences. As mentioned in the previous section, preferences can be used retrospectively to determine what objects were selected. Through their context fields – the goal, problem-space, state, and operator fields – they can also be used to determine what objects were current in the context at the time the object was selected. The following recoding of the example represents that operator *a1* was acceptable in the situation characterized by *g1*, *p1*, and *s1*: that state *s2*

was acceptable in the situation where operator *a1* was additionally selected (it is *a1*'s result); and that operator *a2* was acceptable in the situation where state *s1* has been replaced by state *s2*.

(preference *a1* ^value acceptable ^role operator
^goal *g1* problem-space *p1* ^state *s1*)
(operator *a1* ^name pull-knob)
(preference *s2* ^value acceptable ^role state
^goal *g1* problem-space *p1* ^state *s1* ^operator *a1*)
(preference *a2* ^value acceptable ^role operator
^goal *g1* problem-space *p1* ^state *s2*)
(operator *a2* ^name turn-knob)

As was true of the gaf event earlier, the watch event is now represented without the use of any structures introduced solely for their role in representing episodic knowledge. The explicit structures that are left may actually be larger than some of the previous (this is not true of the gaf example), but they are structures that are already available because of their role in problem solving. This is thus Soar's native form of episodic knowledge. The significance of this is three-fold. First, this is the form of episodic knowledge which is available without positing additional semantics (or apparatus). Second, this form of episodic knowledge will always be around anyway, so it needs to be taken into consideration. Third, because it posits no additional apparatus, it should automatically compose well with the other capabilities in the system.

4.2. Storage

The manner of storage of episodic knowledge is a function of the representation that is used. Components that are directly represented as declarative structures are stored in the actions of productions. The "gaf" example might be stored in a production like the following (or across multiple productions).[4]

```
-->
(occurred <e1> 'event <o1> 'context <c1>)
(object <o1> 'name gaf)
(context <c1> 'experiment 2 'list 1)
```

Though this production is shown without conditions, as described in Section 5, it is necessary (and possible) to add additional conditions to restrict the situations in which such declarative structures are retrieved.

Assumed parts are simply omitted from the actions. However, for the context assumption to work, a representation of the context must appear in the conditions of the production. This doesn't allow the context to be retrieved as an explicit structure, but does constrain the explicit structures to be retrieved only in contexts similar to the ones in which they were learned. The "gaf" example above would be stored as a production like the following one.

```
(context <c1> 'experiment 2 'list 1)
    --> (object <o1> 'name gaf)
```

[4]Symbols enclosed in angle brackets, such as <e1>, are variables.

4.3. Retrieval

Explicit episodic knowledge is retrieved in the same way as are other declarative structures; that is, by a combination of k*-search and ps-search. Omitted components are retrieved by assumption. If the context of occurrence is omitted, then it is assumed to share critical features with the retrieval context. If the predicate is omitted, occurrence is assumed for retrieved information. If actions are omitted, operator retrieval is assumed to denote an action that was executed. If sequence information is omitted, preferences are assumed to denote sequences of operators and states that occurred.

4.4. Use

There has not yet been a great need for episodic knowledge in the tasks that have so far been implemented in Soar. Nonetheless, it has been used in several distinct ways. One way is the use of preferences as the basis for a form of chronological backtracking – a short-term episodic use in which the preferences remain in working memory throughout. Soar normally backtracks in look-ahead search by terminating subgoals that lead to failure. However, there are times when Soar thinks it knows what it is doing – so no look-ahead search is being performed – yet failure still occurs. Backtracking under these circumstances involves examining the acceptable preference for the state at which failure occurred to find out which state was current when the failed state was selected. This prior state is then reselected, and problem solving is continued.

A second use is as the basis for recognition and recall tasks (Rosenbloom, Laird, & Newell, 1987, Rosenbloom, Laird, & Newell, 1988a). In a recognition task the system is presented with a list of items to be memorized. It is then prompted with an item which may or may not be in the list. Its task is to say *yes* if the item was in the list and *no* if it wasn't. A recall task is similar, but instead of being prompted with an item, the system must produce as many of the items in the list as it can, without producing items not in the list. These tasks require episodic knowledge because they ask questions about what happened in the system's past.

A third use is as the basis for the transfer of procedural knowledge. The procedural knowledge that Soar learns can be viewed as really being episodic knowledge about the past behavior of the system. To use this episodic knowledge as procedural knowledge, there is an implicit assumption that what is descriptive of the past is normative for the future. This assumption is maintained until it leads to an error, at which point the system attempts to recover by doing something other than what is directly dictated by its past experience (Laird, 1988).

The issue of errors is actually a key one when native episodic knowledge is used, because, whenever an assumption is made, the possibility for error creeps in. There are four classes of situations that can lead to errors. Some of these are intrinsic in the natur of the world, while others arise because of specific architectural commitments in Soar. The first class of

errors arises because of mistakes in credit assignment. Soar cannot examine the conditions of productions, so when knowledge is retrieved it can only guess as to which aspects of the retrieval context were shared with the context in which the knowledge was learned. This can lead to both errors of commission and omission. Suppose, for example, that the system is winding a watch at the same time it is trying to recall the elements that occurred in list 1 of experiment 2. It will retrieve both "gaf" and "pull-knob then turn-knob". The problem is that there is no a priori reason to assume that one of these events is in the list and that the other is not. In this particular case it might be able to use background or other contextual knowledge to reason that the watch events were not part of the list, but in other cases it may not be so lucky.

The second class of situations arises because of mistakes in context generalization. There is a trade-off between the scope of applicability of knowledge and its utility as episodic knowledge. The more general is the context, the more situations in which the knowledge can be retrieved, and thus be available for use. However, increasing the generality also decreases the ability to discriminate the situations in which the knowledge was originally learned from related situations. This can be seen clearly in the acquisition and use of control knowledge. The more general is the control knowledge, the more search is eliminated, assuming the generalization is correct. However, generality also implies that the knowledge will be retrieved in a variety of contexts, many of which are

only remotely like the one in which the knowledge was learned.

The third class of situations arises because of mistakes in memory attribution. The only examinable structures in Soar are those in its working memory. Such structures could arise from memory retrieval, from intervention by the architecture (the decision procedure), or from perception. Only those that arise from memory retrieval embody episodic knowledge. Normally this shouldn't be a problem because the decision procedure and perceptual systems each create structures in a characteristic fashion: the decision procedure only modifies certain special attributes of goals: perception always adds its structures to special attributes of the state in the top context. However, the possibility remains.

The fourth, and final class of situations can be caused by any of the first three. It occurs because of mistakes in co-occurrence attribution. Such failures occur when the system mistakenly thinks it has previously experienced an event because it has experienced all of its individual pieces, though never all as part of a single event. Suppose, for example, that in one context the system sees a large ball, and in a similar context it sees a green ball. A co-occurrence error occurs if the system thinks that it saw both in the same context, or worse, that it has seen a single large green ball. In the recognition and recall tasks this problem is partially dealt with by assuming that k*-retrievable objects have been experienced, while ps-retrievable and unretrievable objects have not. The rationale is that k*-retrieval, being a limited computational mechanism, has a limited

ability to put things together in novel ways, while ps-retrieval allows arbitrary structures to be created.

4.5. Acquisition

Episodic knowledge is acquired by chunking problem-solving episodes. Though this is somewhat of a tautology for Soar, it is not always as simple as it sounds. The simple case is the acquisition of episodic knowledge about objects generated in subgoals. If such objects are returned as results of their subgoals, then chunks are created which can later be used as episodic knowledge about the objects. The variety of subgoal results – objects, operators, preferences, etc. – leads directly to variety in the episodic knowledge that can be learned.

Under normal circumstances this episodic knowledge is represented in what we have referred to as native form: that is, predicates, contexts, actions, and sequences are represented respectively by chunk existence, production conditions, operators, and preferences. However, if the system monitors its own performance, and creates declarative structures representing what has transpired, then the chunks created for such structures can be used as explicit declarative-form episodic knowledge about what has transpired. In addition, such chunks can be used as native episodic knowledge about the monitoring process itself.

One form of episodic knowledge that cannot be handled this easily is knowledge about what has happened to the system; that is, knowledge

about what the system has perceived rather than knowledge about what the system has generated. The "gaf" example presented above is a typical perceptual event. The episodic knowledge to be acquired is about the perception of "gaf" in a particular context (experiment 2, list 1). As described in Section 1, Soar's input mechanism attaches perceptual information to the state in the top problem-solving context. In order for information about the event to be stored into long-term memory by chunking, an internal episode must be generated in which this perceptual information is used.

For perceptual events it is relatively easy to acquire a form of episodic knowledge akin to a familiarity test. The system must simply chunk over a subgoal in which it examines a representation of the perceptual event and the context, and generates as a subgoal result an occurred predicate covering them (Rosenbloom, Laird, & Newell, 1987). A familiarity chunk for this example might look like one of the following two productions, depending on whether the context is explicit or not.

```
(object <o1> ^name gaf)
(context <c1> ^experiment 2 ^list 1)
-->
(occurred <e1> ^event <o1> ^context <c1>)

(object <o1> ^name gaf)
(context <c1> ^experiment 2 ^list 1)
-->
(occurred <e1> ^event <o1>)
```

In the systems so far implemented, context is actually ignored in the learning of episodic knowledge. By having no explicit representation of

context in either the conditions or the actions of the chunks, the context is effectively the entire history of the system. In this form, the "gaf" familiarity chunk looks like the following:

```
(object <o1> ^name gaf)
-->
(occurred <e1> ^event <o1>)
```

Familiarity chunks allow the determination of whether an event has occurred before in a particular context. Whenever a representation of the event appears in working memory along with a representation of the context, an occurred predicate will be retrieved for them. Familiarity chunks can thus support performance in recognition tasks, where the task is to determine whether a presented object has been seen before.

What familiarity chunks do not directly support is the retrieval of events that occurred in a particular context. For an event to be retrieved by the execution of a chunk, the event must be stored in the actions of the chunk, and not tested in its conditions. A retrieval chunk for the "gaf" example should look something like the following (with an assumed context and predicate):

```
(context <c1> ^experiment 2 ^list 1)
-->
(object <o1> ^name gaf)
```

For such productions to be learned by Soar, they must be created by chunking over some form of problem solving. But chunking is not an indefinitely flexible mechanism. A chunk's actions are always based on

the results of a subgoal, and its conditions are always based on a dependency analysis of the results. This immediately imposes two constraints on the nature of the problem solving that can underly the acquisition of retrieval chunks.

1. For the event to appear in the actions of a chunk, it must be generated as a result of a subgoal.

2. For the event to not appear in the conditions of the chunk, the subgoal results must not depend on an examination of the event.

The first constraint is relatively easy to meet: for example, by creating a copy of the perceptual event in a subgoal, and returning the copy as a result. However, attempting to meet both constraints at once leads to the *data chunking problem*: if the result is based on examining the object to be learned, then the conditions of the chunk will also test the object, allowing it to only be retrieved when it is already available. For example, using the copying strategy for the "gaf" example would lead to a chunk like the following:

```
(context <c1> ^experiment 2 ^list 1)
(object <o1> ^name gaf)
-->
(object <o1> ^name gaf)
```

In contrast to the desired retrieval chunk, this one tests that "gaf" is already in working memory before it will retrieve it. So it doesn't do the job.

The solution to the data chunking problem is to separate the result

generation process from the use of the perceptual event. Result generation must be based on what the system already knows, rather than on the perceptual event. One approach involves assembling the result from components that the system can already retrieve (Rosenbloom, Laird, & Newell, 1987). For example, if the letters "g", "a", and "f" are retrievable, then "gaf" can be generated by retrieving and assembling them. This is a syntactic compositional process which may or may not respect any specific semantic rules in performing the assembly. Another approach is to start with the context and to chain through a sequence of productions which form a pre-existing, though possibly indirect, link between the context and the event (Rosenbloom, 1988). For example, (object *o1* ^name gaf) can be generated if (context *c1* ^experiment 2 ^list 1) is already in working memory, and if the following two retrieval chunks exist.

```
(context <c1> ^experiment 2 ^list 1)
-->
(object <o1> ^name fem)

(object <o1> ^name fem)
-->
(object <o1> ^name gaf)
```

Either approach requires the system to start out with a set of primitive elements that can be generated. Other more complex structures can then be built up out of compositions of these primitive elements. For the work on recognition and recall, the system was initialized with the ability to generate the 26 letters. Conceivably, Soar could have been initialized with an even lower level of primitives, such as simple lines, curves, and points,

from which it would construct the letters. It could also have been initialized with more meaningful primitives, such as the primitive ACTs in Conceptual Dependency Theory (Schank, 1975) or the epistemological primitives in KL-ONE (Brachman, 1979).

Though the perceptual event cannot be used directly in the generation process, it is still used in two critical ways. The first is as the basis of a goal test for the generation process. The generation process can conceivably return any event that it can either assemble or chain to, so a goal test is necessary to determine when the desired event has been generated. The straightforward approach of comparing the perceptual and generated events does not work. Instead, it leads to a secondary version of the data chunking problem in which the comparison causes tests of the perceptual event to appear in the conditions of the chunk.

To avoid this secondary data chunking problem, the goal test is based on a familiarity chunk for the perceptual event rather than directly on the event. Given a familiarity chunk for the perceptual event, the generation goal test is satisfied when a familiar but unretrieved event is generated. The test of familiarity guarantees that the event has been seen in the current context (or a similar one). If the event is unretrieved, it is one that the system has not previously learned to generate. This test is somewhat overgeneral in that it can't guarantee that the generated event is a copy of the current perceptual event. However, at worst it will only generate a different event that is familiar in the same context. If this

happens, it is always possible to try again to generate an event that corresponds to the input event. The use of a familiarity chunk as the goal test for the generation process makes this is a generate-recognize approach to recall (see, for example, Watkins & Gardiner, 1979), though focused on the acquisition phase rather than the retrieval phase.

The second way that the perceptual event is used is as the basis for controlling the search through the space of events that can be generated. The goal test determines the correctness of the result, but does not affect the efficiency of the search. Using the perceptual event as control knowledge makes the search tractable, potentially removing all backtracking, without affecting the correctness of the result. Thus the result technically does not depend on such control knowledge. The bottom line is that the use of search control knowledge can speed up performance without introducing additional conditions into chunks (recall from Section 1 that chunking does not bactrace through the use of desirability preferences by the decision procedure). Of secondary importance is that the use of the perceptual event as search control increases the likelihood that the first event generated will correspond to the perceptual event rather than to another familiar but unretrieved event.

When the generation, goal testing, and control process are all put together, a retrieval chunk can be learned that is identical to the one that was desired. Context can be treated in the same ways that it is for

familiarity chunks. It can be explicit (in the actions), assumed (in the conditions), or completely ignored (nowhere). The systems so far implemented have ignored context.

One consequence of this approach to the acquisition of retrieval chunks is that it forces information storage to be based on an understanding process. The understanding may be only of syntax (surface structure), or it may be of a deeper semantic (deep structure) nature, but without it, learning will not occur. There is no simple assignment operation – such as the SETQ operation in Lisp – that allows an unanalyzed structure to be stored in long-term memory. A second consequence is that the understanding process must be a reconstructive – or analysis-by-synthesis – process (Bartlett, 1932, Neisser, 1967), in which events are reconstructed in terms of known structures. A third consequence is that the storage process is semantically penetrable. Other knowledge can potentially alter the reconstruction process, and thus what is stored, leading to generalization and other forms of bias in the memory structures that are stored.

4.6. Summary

Soar can represent, store, retrieve, use, and acquire episodic knowledge. However, the situation is nowhere near as clean and simple as it was for procedural knowledge. In fact, if we were to sit down to design a capability for episodic knowledge from scratch, with no constraints, we would be unlikely to design it as currently embodied by Soar. What the

architecture most directly supports is native episodic knowledge about structures generated by the system itself. Such knowledge is represented, stored, retrieved, and acquired without requiring additional cognitive effort. However, the assumptions required to use such knowledge can lead to errors. By increasing cognitive effort, more explicit forms of episodic knowledge can be acquired that require fewer assumptions for use, and thus hopefully lead to fewer errors. Such structures are not terribly dissimilar to the structures used in other episodic memory proposals, such as Scripts (Schank and Ableson, 1977) and E-Mops (Kolodner, 1985).

The situation is even more complicated for episodic knowledge about perceptual events. Three features of the architecture yield strong constraints on how the knowledge is acquired.

1. Chunk actions are based on subgoal results.
2. Chunk conditions are based on the supergoal structures upon which the results depend.
3. Perceptual information arrives in the top goal.

Together these features force a reconstructive approach to knowledge acquisition. Though this approach is considerably more complicated than simple verbatim storage of what has transpired, it does have a number of promising properties.

Given the overall picture of episodic knowledge, as relatively complicated and messy, it is important to ask whether this signals a need for modification of the architecture. One key question is the appropriateness

of the levels of support and constraint that are provided by the architecture. For psychology, this is a matter of the extent to which the level of support models human capabilities, and the level of constraint models human limitations. For AI, this is a matter of whether the system achieves an appropriate level of episodic functionality. A second key question is whether there are more appropriate mechanisms – for either definition of "appropriate" – for the support of episodic knowledge which could be integrated cleanly into the architecture. Providing detailed answers to these two key questions remains for future work.

5. Declarative Knowledge

Declarative knowledge is knowledge about what is true in the world. Examples include the facts that dogs have four legs and that Fido is a dog. Declarative representation comes in many forms, such as natural language, diagrams, maps, charts, tables, and graphs. This is also the area with which logic is classically concerned. A logic has a syntax specifying the form that statements take, and a semantics which relates logical statements to an abstract conceptualization of the world. For first order predicate calculus (FOPC) the syntax is based on constants, variables, predicates, connectives ($\wedge$, $\vee$, $\neg$, and $\supset$), and quantifiers ($\forall$ and $\exists$). The mapping between the syntax and the semantics, called the interpretation, is used to determine the truth of statements expressed in the syntax of the logic.

In this section we will be primarily concerned with the syntactic side of

declarative knowledge, taking advantage of logic's privileged role in AI as a language for representing declarative knowledge. We will examine how declarative knowledge is represented (its syntax), stored, retrieved, used, and acquired. On the issue of semantics, we will assume that the meaning of a structures is determined by a combination of two factors: its relationship to the outside world, as mediated by the perceptual and motor systems, and how it is used by internal processes.

As with episodic knowledge, the main challenge will be in the acquisition of declarative knowledge. The data chunking techniques described earlier will be extended to handle declarative representations.

5.1. Representation

As with episodic and procedural knowledge, there are several different ways that declarative knowledge can be represented in Soar.[5] The most flexible and general approach is to represent each syntactic component – whether it be a constant, variable, predicate, connective, or quantifier – as an object. The details of exactly how this is done are not crucial, but the general flavor should be clear from the following example which shows a statement in FOPC and how it could be translated into a set of objects in Soar.

---

[5] In this section we focus on derivation-based representations for logic. Other representations are possible, such as validity-based techniques – see, for example, (Polk & Newell, 1988) and (Polk, Newell, & Lewis, 1989) for research on mental models in Soar. These have somewhat different properties, but much of the discussion would remain the same.

$\forall x\,[P(x) \supset Q(x)]$

(quantifier *o1* ^name forall ^variable *o1* ^body *b1*)
(variable *o1* ^name x)
(connective *b1* ^name implies ^antecedent *o1* ^consequent *o2*)
(predicate *o1* ^name p ^argument *o1*)
(predicate *o2* ^name q ^argument *o1*)

For simple statements containing no quantifiers or variables, no predicates with more than two arguments, and no connectives except for $\wedge$, there is a simpler native representation that takes direct advantage of Soar's object structure. Constants are represented as objects, predicates as attributes, and conjunction as simultaneous occurrence. Here's a simple example about Fido.

Category(Fido, Dog) $\wedge$ Alive(Fido)

(object *o1* ^name fido ^category *o2* ^alive)
(object *o2* ^name dog)

This native representation is more succinct than the one above, but in exchange it lacks expressibility.

Another variation is to use productions as a representation for a subclass of implications. A production can represent a universally quantified implication in which the antecedent is a conjunction of predicates, and the consequent is an existentially quantified conjunction of predicates.

$\forall(x_1,\ldots,x_n)[C_1(x_1,\ldots,x_n) \wedge \ldots \wedge C_j(x_1,\ldots,x_n) \supset$
$\exists(y_1,\ldots,y_m)\,[A_1(x_1,\ldots,x_n,y_1,\ldots,y_m) \wedge \ldots \wedge A_k(x_1,\ldots,x_n,y_1,\ldots,y_m)]]$

The existential quantifier in the consequent arises from the ability to create new objects when variables appear in actions that are not in

conditions. A form of negation as failure is also available in the antecedent which allows a predicate to be assumed false unless it is explicitly known to be true (that is, available in working memory).

Implications encoded as productions are non-examinable and can only be used to forward chain. We will not discuss them further here. Instead we will focus on examinable structures of either of the first two types.

5.2. Storage

Storage of declarative knowledge is straightforward. As with declaratively represented procedural and episodic knowledge, declarative knowledge is stored in the actions of productions.

5.3. Retrieval

Declarative knowledge is retrieved by k*-search and ps-search. If the knowledge is stored directly in productions, k*-search can retrieve it, otherwise ps-search is required. Hopefully, by this point, this is obvious. However, less obvious is what the context should be for retrieval of declarative knowledge. The retrieval contexts for both procedural and episodic knowledge are straightforward. An element of procedural knowledge is retrieved when it may be needed to produce behavior. An element of episodic knowledge is retrieved when a context is established that is similar to the one in which the episode occurred (at least if the context is tested in production conditions rather than stored in production actions). In contrast, declarative knowledge is by its essence not

associated with a particular context. The knowledge is true, independent of context, and should be usable in any context requiring it.

On the other hand, if declarative knowledge is stored with no context – that is, with a null set of production conditions – the knowledge will not only be retrievable in all contexts, it will in fact be retrieved in all contexts, swamping the system with true but irrelevant information. One approach to controlling the retrieval of declarative knowledge is to use the connectedness among facts as a co-relevance heuristic. This is often implemented by the mechanism of spreading activation, which retrieves facts close to those that are already retrieved (Collins & Loftus, 1975, Anderson, 1983). Another approach is to control the retrieval of declarative knowledge by providing a partial description of the knowledge to be retrieved (Norman & Bobrow, 1979). The partial description then delineates the set of things which appear to be relevant.

The approach that we have taken is to store declarative knowledge in a discrimination network that allows retrieval of objects by partial description (Rosenbloom, Laird, & Newell, 1988a). Given any partial description, a single object is retrieved along with the facts about it. The construction of this discrimination network is discussed below, under acquisition.

One last important aspect of the retrieval of declarative knowledge concerns the basis for believing that structures retrieved from long-term

memory represent true facts about the world. This belief must be based on the implicit assumption that the system knew what it was doing during the episode in which the structures were acquired. In other words, the system must trust its past behavior. This is one form of assumption that cannot be completely avoided by adding more explicit structure. Even if explicit true-in-world annotations are added to all structures representing true facts about the world, the system must still trust that in the past it only added such annotations when the facts were true. It might be more "careful" about adding such annotations than it is about adding structures to working memory in general, but since there is no oracle for truth, it is still assuming that these annotations were added correctly. This assumption that the annotations are true is of the same type as the original one. It may localize the assignment of trust, but can not completely eliminate it.

5.4. Use

Declarative knowledge has a multitude of uses. It can be used to describe procedures so that they can be reasoned about, or followed interpretively (from which native procedures can be compiled). It can be used to explicitly describe episodes. It can be used to describe knowledge whose function is not yet clear. It can be used as the basis for memorization tasks. A complete list would go on considerably longer, but this gives a sampling of typical uses.

5.5. Acquisition

The acquisition of declarative knowledge has much in common with the acquisition of episodic knowledge. Declarative knowledge that is generated as the result of a problem solving episode is directly acquired by chunking, with a retrieval context that corresponds to those elements of the situation on which creation of the knowledge depended. Likewise, the acquisition of perceptually originating declarative knowledge utilizes the data chunking solution described in Section 4.5. Though, to go beyond the types of structures acquired in the research on episodic knowledge, Soar was initialized with primitive elements for the 26 letters, plus a set of primitive attributes (isa, has, color, response, letter1, letter2, letter3, letter4, letter5, letter6, letter7, letter8, letter9, letter10).[6] Using these primitives, facts are represented by attributes relating named objects. For example, Isa(Fido, Dog) is represented by the following structures.

```
(object <f> ^letter1 <f1> ^letter2 <f2> letter3 <f3>
    ^letter4 <f4> ^isa <d>)
(letter <f1> ^name f)
(letter <f2> ^name i)
(letter <f3> ^name d)
(letter <f4> ^name o)
(object <d> ^letter1 <d1> ^letter2 <d2> letter3 <d3>)
(letter <d1> ^name d)
(letter <d2> ^name o)
(letter <d3> ^name g)
```

This is a variation on the native representation described in Section 5.1. The primary difference is that names, which were unanalyzable atoms in

[6] In future work we will be examining how to loosen up the requirement that attributes be pre-existing primitives as well as investigating different levels of primitive elements.

the earlier representation, have been expanded out to where their internal structure is open for examination and creation.

There are a number of ways in which the acquisition of declarative knowledge is more complicated than the acquisition of episodic knowledge in Soar. We have so far isolated three additional issues that must be resolved in the acquisition of perceptually originating declarative knowledge. (1) How is a discrimination network to be acquired that can control the retrieval of declarative knowledge? (2) How is knowledge about objects acquired incrementally? (3) How do chunks store the components of an object?

As mentioned earlier, utilizing the data chunking solution alone results in the acquisition of context-free declarative knowledge.[7] Acquiring a discrimination network thus requires an augmentation of the basic data chunking solution. Abstractly, the approach is to modify the simple memorization strategy underlying data chunking so that the acquisition of new knowledge involves relating the new knowledge to what is already known. If in the process of establishing relations, an explanation is created as to why the new knowledge is different from similar existing knowledge, this should lead to discrimination. Similar processing could lead to generalization, or other alterations of the new knowledge prior to

[7]There is a corresponding, but not identical, issue for perceptually originating episodic knowledge which has not yet been addressed: how the situational context is incorporated into chunk conditions.

its being stored (Anderson, 1986, Rosenbloom, 1988).

The current implementation in Soar uses such a strategy to perform object-centered discrimination. Given a new fact, such as Isa(Fido, Dog), the system uses the features of Fido to see what object is retrieved from the discrimination network. If Fido is retrieved, no discrimination is necessary. If some other object, such as Fred, is retrieved, Fido's features are compared with Fred's to find a difference, such as the letter "d" in the third position of the name.[8] This difference is then used as the justification for generating a new symbol representing Fido, and for rejecting Fred as the object to be retrieved. If there is more than one difference, one is picked indifferently.

By thus loosening the prohibition against examining perceptual knowledge during result generation, discriminating conditions are added to retrieval chunks which control when the acquired knowledge is retrieved. Schematically, the production resulting from this process looks like the following.

Retrieved(g35) $\wedge$ $\neg$ Rejected(g35) $\wedge$ letter3(g35, e)
$\wedge$ Perceived(p) $\wedge$ letter3(p, d) --> Reject(g35) $\wedge$ Retrieve(g37) (1)

This production says that if there is a retrieved, but not yet rejected, object with symbol g35 and an "e" as its third letter, and the perceived object's third letter is "d", then reject the retrieved object and create a

[8] In the current implementation, discrimination is always based on features of object names, rather than on other facts known about the object.

new symbol (g37) for the perceived object. Each such retrieval production forms one link in the discrimination network that is constructed as new objects are perceived. This discrimination network supports k*-retrievability – that is, retrievability by a combination of production firings within a single elaboration phase – rather than the k-retrievability that is possible for knowledge stored with no context.

Consider what happens when the following three facts are learned in sequence, ignoring for now all of the learning except for the creation of the discrimination network.

Isa(Fred, Cat)
Isa(Fido, Dog)
Isa(Carl, Dog)

First, Fred is processed. Because no object has been learned previously, no discrimination is necessary, and the only action to be taken is the creation of a new symbol for Fred. This results in the acquisition of a production which generates the symbol for Fred if no object has already been retrieved.

¬ Retrieved() --> Retrieve(g35) (2)

Second, Cat is processed. Given the features of Cat, the symbol for Fred (g35) is retrieved by production 2. As described below, the information about Fred is cued off of Fred's symbol, so the retrieval of g35 leads to the retrieval of what is known about Fred. Once this knowledge is retrieved, Cat is discriminated from Fred. The system chooses indifferently one of the discriminating letters of the objects' names – in this case the second letter – yielding the following production.

Retrieved(g35) ∧ ¬ Rejected(g35) ∧ letter2(g35, r)
∧ Perceived(*p*) ∧ letter2(*p*, a) --> Reject(g35) ∧ Retrieve(g36) (3)

Third, Fido is processed. Fred is retrieved and discriminated from Fido by the third letter, yielding production 1, above. Fourth, Dog is processed. Once again, Fred is retrieved, and the discrimination is again based on the third letter, yielding the following production.

Retrieved(g35) ∧ ¬ Rejected(g35) ∧ letter3(g35, e)
∧ Perceived(*p*) ∧ letter3(*p*, g) --> Reject(g35) ∧ Retrieve(g38) (4)

Fifth, Carl is processed. This time Fred is retrieved and then immediately rejected by production 3, which also retrieves Cat. Carl is then discriminated from Cat by the third letter, yielding the following production.

Retrieved(g36) ∧ ¬ Rejected(g36) ∧ letter3(g35, t)
∧ Perceived(*p*) ∧ letter3(*p*, r) --> Reject(g36) ∧ Retrieve(g39) (5)

Sixth, Dog is processed. Fred is retrieved, and then immediately rejected by production 4, which also retrieves the symbol for Dog. Because there is no mismatch between the perceived and retrieved objects, no discrimination is necessary, no new symbol is generated, and no new production is created. At this point the discrimination network has the shape shown in Figure 5-1.

Given a partial specification of an object name, the symbol for the object whose name matches most closely – according to the structure of the discrimination network – is retrieved. The knowledge associated with the object is acquired with a retrieval context consisting of the object's symbol. As with episodic knowledge, this occurs by first acquiring a

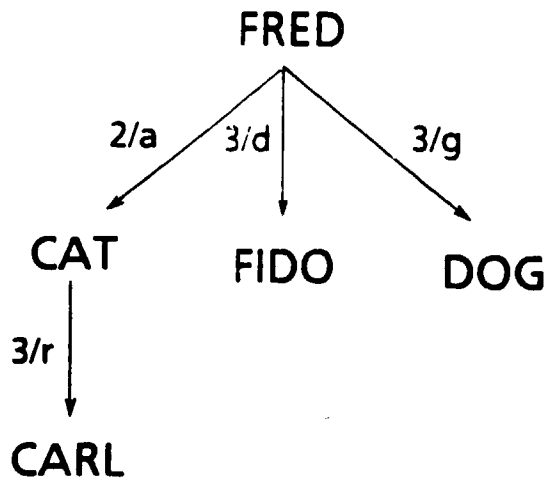


Figure 5-1: Discrimination network acquired from sequence of facts.

familiarity chunk for the new knowledge, and then acquiring a retrieval chunk which depends on the object's symbol.

Unfortunately, this approach depends on having all of the knowledge about the object available at once, which raises the second issue: how to incrementally acquire knowledge about objects. If the familiarity chunk must recognize the entire object, as it does in the previously published approaches, no learning can occur about an object until all of the knowledge about it is available. Conceptually, the solution to this problem is straightforward. Data chunking is applied to each fragment of an object individually. The processes of familiarization and generation are performed independently for each letter of the object's name, and for each fact about the object. As an example, the individual familiarity and retrieval productions for the first letter of Fred's name look like the

following.

letter1(g35, f) --> Familiar(letter1(g35, f)) (6)

Retrieved(g35) ∧ ~ Rejected(g35) --> letter1(g35, f) (7)

The implementation of this solution allows new facts to be acquired about known objects, for example, that Color(Fido, Red). However, what is given up in going with this solution is the ability to use familiarity chunks to directly perform recognition tasks. Object recognition must now be based on multiple productions. The obvious way to do this is to sort the object through the discrimination network. If the new object is the same as the one stored in the node at which discrimination ends, then it is recognized.

The third issue is how references to learned objects should be stored in retrieval chunks. For primitive objects, such as the letters, the answer is simple. The value is stored directly in the actions of the retrieval chunk, as shown in production 7. However, for values that are objects, the situation is more complicated and leads to a sequenced pair of discriminations, similar to the approach taken by EPAM (Feigenbaum & Simon, 1984).

Suppose the system has already learned about the object bej, a nonsense trigram, and that the retrieval cue associated with bej is its first letter (b). Then suppose that the system is presented with a paired associate (a stimulus-response pair) in which bej is the response, for example

Response(gaf. bej). This pair would be represented as follows.

(object *s* ^letter1 *s1* ^letter2 *s2* ^letter3 *s3* ^response *r*)
(letter *s1* ^name g)
(letter *s2* ^name a)
(letter *s3* ^name f)
(object *r* ^letter1 *r1* ^letter2 *r2* ^letter3 *r3*)
(letter *r1* ^name b)
(letter *r2* ^name e)
(letter *r3* ^name j)

If g41 is the symbol for gaf and g42 is the symbol for bej, the obvious retrieval production to create for this pair (ignoring for now the retrieval productions for the object's names) is the following.

Retrieved(g41) ∧ ¬ Rejected(g41) --> response(g41, g42) (8)

However, the rule that is actually created will have an additional condition which tests bej's retrieval cue (b).

Retrieved(g41) ∧ ¬ Rejected(g41) ∧ Letter1(g42, b)
--> response(g41, g42) (9)

This happens because the retrieval cue is examined in order to retrieve g42 from the discrimination network. Therefore, the appearance of g42 in the chunk's actions leads to the cue appearing in the chunk's conditions. Such a retrieval production cannot support performance in paired-associate tasks, where after studying a list of stimulus-response pairs, the subject must generate responses when given just the corresponding stimuli.

One solution to this problem is to include a (partial) description of the value object in the retrieval production rather than the object itself. If the description is created anew, by data chunking, from the information provided in the paired associate, no additional retrieval cues get

;

incorporated. With this change, the following chunk is learned.

Retrieved(g41) ∧ − Rejected(g41) --> response(g41, letter1(-, b)) (10)

Given such a chunk (or set of chunks), the paired associate task can be performed by using the stimulus features to retrieve its symbol (g41), using the stimulus symbol to retrieve a (possibly partial) description of the response (letter1(-, b)), using the description of the response to retrieve the response symbol (g42), and then using the response symbol to retrieve the information about the response (bej).

Using this strategy the task now requires two independent retrievals from the discrimination network − one to retrieve the stimulus symbol and one to retrieve the response symbol. This is similar to the double discrimination performed by EPAM during paired associate tasks (Feigenbaum & Simon, 1984). However, it occurs in Soar not because of a need to match the data, but as a means of enabling responses to be retrieved from stimuli in the absence of any additional response cues.

5.6. Summary

The representation, storage and use of declarative knowledge are relatively straightforward in Soar. The architecture provides some support for these capabilities through the primitive attribute-value representation, the ability to store declarative structures in productions, and the ability to examine declarative structures in working memory. The architecture does not enforce a fixed semantics for declarative knowledge, nor does it provide default inference mechanisms that automatically

generate structures representing knowledge that is implied by its existing structures. Both of these, if part of the architecture, would imply excessive rigidity in how the system could behave. To the extent they are needed, provision should be made by adding knowledge and problem spaces on top of the fixed architecture.

The acquisition and retrieval of declarative knowledge is considerably less straightforward. Chunking is provided as a means for acquiring declarative knowledge, but the problem solver must be put through a number of contortions for the system to acquire the appropriate chunks from externally provided knowledge. In addition to the constraint imposed by the data chunking problem, a related architectural constraint restricts how known objects can be used in the acquisition of new knowledge. The solution to the problem posed by this constraint leads to an approach which is increasingly like the discrimination network structure of the EPAM model of memory.

Retrieval of declarative knowledge is supported by production firing, but considerable additional complexity arises from the need to constrain retrieval to what might be relevant. The discrimination network we have employed provides one approach to this. Though the approach currently seems somewhat ad hoc, the abstract characterization of the process as relating the new knowledge to existing knowledge, leads to the hope that it will eventually be placed on a principled footing.

Further work is clearly called for in developing and evaluating the acquisition and retrieval mechanisms that have been proposed (and implemented).

6. Conclusions

Taking an architecture seriously means living within its constraints and using what support it provides, at least until it is clear that the architecture must be modified. In this chapter we have taken a step towards evaluating the level of support and constraint which the Soar architecture provides for the knowledge level, a concept that is closely related to the idea of general intelligence. This helps us to understand the extent to which the architecture's current levels of support and constraint are adequate for achieving general intelligence. It also provides an alternative way of viewing Soar in which its architectural mechanisms are subjugated to their role in supporting knowledge. This complements other efforts that view Soar as a set of mechanisms (Laird, Newell, & Rosenbloom, 1987), a hierarchy of meta-levels (Rosenbloom, Laird, & Newell, 1988b), a hierarchy of cognitive levels at different time scales (Newell, 1989, Rosenbloom, Laird, Newell, & McCarl, 1989), a physical symbol system (Newell, Rosenbloom & Laird, 1989), and a general goal-oriented system (Rosenbloom, 1989).

Our particular focus in this step has been on how the Soar architecture supports and constrains the representation, storage, retrieval, use, and acquisition of three pervasive forms of knowledge: procedural, episodic,

and declarative. The analysis reveals that Soar adequately supports procedural knowledge – to some extent it was designed for this – but that there are still significant questions about episodic and declarative knowledge. These questions arise primarily because of consequences of the principle source of constraint in Soar, the fact that all learning occurs via chunking. Chunking can support the acquisition of episodic and declarative knowledge, but in so doing it imposes significant requirements on how the problem solving underlying this acquisition proceeds. These requirements amount to architecturally-derived hypotheses about how learning occurs. We have reported here some new results that elaborate on these hypotheses in the acquisition of declarative knowledge, but considerable future work is still called for in both the development and testing of these hypotheses.

One obvious question at this point is why not just add new architectural mechanisms that directly support the acquisition of episodic and declarative knowledge? Assuming that appropriate mechanisms could be developed, there are still at least two critical reasons not to rush into adding them to the architecture. The first reason is that the integration of new mechanisms into an existing architecture can have major consequences. An integrated architecture is more than just a collection of useful mechanisms. It must be constructed so that its mechanisms compose appropriately with each other. The number of potential interactions that need to be worried about increases rapidly –

exponentially, if there can be interactions among all possible subsets – with the number of mechanisms. Frequently, the addition of a perfectly reasonable new mechanism will cause strongly dysfunctional behavior in others of the existing mechanisms. Though there are times when an architectural addition is absolutely required, and a research effort must be engendered to get the interactions right (as recently occurred for perceptual-motor behavior in Soar (Wiesmeyer, 1988)), almost always a conservative strategy is what is required.

The second reason is that rushing to add new mechanisms discourages learning about the limits of the existing mechanisms, and their combinations. This is essential to understanding the scope and limits of the architecture. It is also essential to discovering the deeper, nonobvious consequences of the architecture. If we had jumped to add new learning mechanisms to Soar, we would never have discovered how the current mechanisms inherently imply a reconstructive learning strategy. The discovery of such nonobvious consequences is some of the most interesting research that can be done with architectures.

This being said, much additional work is still needed. One issue to be addressed is the origins of the bootstrap knowledge that allows new procedural, episodic, and declarative knowledge to be acquired. The acquisition of perceived procedural knowledge requires the existence of an interpreter for the knowledge. The acquisition of perceived episodic and declarative knowledge requires a set of pre-existing primitive elements plus

the knowledge about how to familiarize, discriminate, and construct object representations. It appears necessary to add some of this to the architecture, such as the ability to generate a set of primitive elements. Other parts may just be specific instances of more general capabilities, which of course must themselves be either innate or learned. For example, the interpreter for procedural knowledge may be just an instantiation of a more general comprehension process. The same may also be true of the discrimination and construction processes for declarative and episodic knowledge. The current implementation does not quite look like this, and architectural changes may be required before it does, but this is one promising path to pursue.

Finally, a number of additional steps must still be taken before the relationship of the Soar architecture to the knowledge level is completely tied down. The most important missing aspect is the relationship between Soar's mechanisms and the principle of rationality. The key issue is how its architectural mechanisms, such as its decision procedure and subgoal generator, allow Soar to approximate rationality even under the constraints of its being a physical system with time and space bounds. We have commented briefly on how chunking increases Soar's ability to bring knowledge to bear under real-time constraints, but much more is left to be done.

References

Anderson, J. R. (1983). A spreading-activation theory of memory. *Journal of Verbal Learning and Verbal Behavior, 22*, 261-295.

Anderson, J. R. (1986). Knowledge compilation: The general learning mechanism. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.). *Machine Learning: An Artificial Intelligence Approach. Volume II.* Los Altos, CA: Morgan Kaufmann Publishers, Inc.

Bartlett, F. C. (1932). *Remembering: A Study in Experimental and Social Psychology.* Cambridge, Eng.: Cambridge University Press.

Bell, C. G. & Newell, A. (1971). *Computer Structures: Readings and Examples.* New York: McGraw-Hill.

Brachman, R. J. (1979). On the epistemological status of semantic nets. Findler, N. V. (Ed.). *Associative Networks.* New York, Academic Press.

Collins, A. M., & Loftus, E. F. (1975). A spreading-activation theory of semantic processing. *Psychological Review, 82*, 407-428.

Feigenbaum, E. A., & Simon, H. A. (1984). EPAM-like models of recognition and learning. *Cognitive Science, 8*, 305-336.

Golding, A., Rosenbloom, P. S., & Laird, J. E. (1987). Learning general search control from outside guidance. *Proceedings of IJCAI-87.* Milan.

Kolodner, J. L. (1985). Memory for experience. In G. Bower (Ed.),

Psychology of Learning and Motivation. Volume 19. New York. NY: Academic Press.

Laird. J. E. (1983). *Universal Subgoaling.* Doctoral dissertation. Carnegie-Mellon University. (Available in Laird. J. E.. Rosenbloom. P. S.. & Newell. A. *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies.* Hingham. MA: Kluwer. 1986).

Laird. J. E. (1988). Recovery from incorrect knowledge in Soar. *Proceedings of AAAI-88.* St. Paul.

Laird. J. E., and Newell. A. (1983). A universal weak method: Summary of results. *Proceedings of IJCAI-83.* Karlsruhe.

Laird. J. E.. Swedlow. K. R.. Altmann. E.. Congdon. C. B.. & Wiesmeyer. M. (1989). *Soar User's Manual: Version 4.5* .

Laird. J. E.. Newell. A.. & Rosenbloom. P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence. 33.* 1-64.

Laird. J. E.. Yager. E. S.. Tuck. C. M.. & Hucka. M. (1989). Learning in tele-autonomous systems using Soar. *Proceedings of the NASA Conference on Space Telerobotics.* Pasadena. CA. In press.

Lewis. R. L.. Newell. A.. & Polk. T. A. (1989). Toward a Soar theory of taking instructions for immediate reasoning tasks. *Proceedings of the 11th Annual Conference of the Cognitive Science Society.* Ann Arbor. MI. In press.

Neisser, U. (1967). *Cognitive Psychology.* New York: Appleton-Century-Crofts.

Newell, A. (1981). The knowledge level. *AI Magazine, 2,* 1-20.

Newell, A. (1989). *Unified Theories of Cognition.* Cambridge, MA: Harvard University Press. In press.

Newell, A., Rosenbloom, P. S., & Laird, J. E. (1989). Symbolic architectures for cognition. In M. I. Posner (Ed.), *Foundations of Cognitive Science.* Cambridge, MA: Bradford Books/MIT Press. In press.

Norman, D. A., & Bobrow, D. G. (1979). Descriptions: An intermediate stage in memory retrieval. *Cognitive Psychology, 11,* 107-123.

Polk, T. A. & Newell, A. (1988). Modeling human syllogistic reasoning in Soar. *Proceedings of the 10th Annual Conference of the Cognitive Science Society.* Montreal.

Polk, T. A., Newell, A., & Lewis, R. L. (1989). Toward a unified theory of immediate reasoning in Soar. *Proceedings of the 11th Annual Conference of the Cognitive Science Society.* Ann Arbor, MI. In press.

Rosenbloom, P. S. (1988). Beyond generalization as search: Towards a unified framework for the acquisition of new knowledge. G. F. DeJong (Ed.), *Proceedings of the AAAI Symposium on Explanation-Based Learning.* Stanford, CA: AAAI.

Rosenbloom, P. S. (1989). A symbolic goal-oriented perspective on connectionism and Soar. In R. Pfeifer, Z. Schreter, F. Fogelman-Soulie, & L. Steels (Eds.), *Connectionism in Perspective*. Amsterdam: Elsevier. In press.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (1987). Knowledge level learning in Soar. *Proceedings of AAAI-87*. Seattle.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (1988). The chunking of skill and knowledge. In B. A. G. Elsendoorn & H. Bouma (Eds.), *Working Models of Human Perception*. London: Academic Press.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (1988). Meta-levels in Soar. In P. Maes & D. Nardi (Eds.), *Meta-Level Architectures and Reflection*. Amsterdam: North Holland.

Rosenbloom, P. S., Laird, J. E., Newell, A., & McCarl, R. (1989). A preliminary analysis of the foundations of the Soar architecture as a basis for general intelligence. In D. Kirsh & C. Hewitt (Eds.), *Foundations of Artificial Intelligence*. Cambridge, MA: MIT Press. In press.

Schank, R. C. (1975). *Conceptual Information Processing*. Amsterdam: North Holland.

Schank, R. and Ableson, R. (1977). *Scripts, Plans, Goals and Understanding*. Hillsdale, NJ: Lawrence Erlbaum.

Steier, D. M., Laird, J. E., Newell, A., Rosenbloom, P. S., Flynn, R.,

Golding, A., Polk, T. A., Shivers, O. G., Unruh, A., & Yost, G. R. (1987). Varieties of Learning in Soar: 1987. P. Langley (Ed.), *Proceedings of the Fourth International Workshop on Machine Learning*. Los Altos, CA: Morgan Kaufmann Publishers, Inc.

Tulving, E. (1983). *Elements of Episodic Memory*. New York: Oxford University Press.

Watkins, M. J., & Gardiner, J. M. (1979). An appreciation of generate-recognize theory of recall. *Journal of Verbal Learning and Verbal Behavior, 18*, 687-704.

Wiesmeyer, M. (1988). *Soar I/O Reference Manual, Version 2*.

Yost, G. R. (1987). *TAQ: Soar Task Acquisition System, Version 2*.

Yost, G. R., & Newell, A. (1988). Learning New Tasks in Soar. Unpublished.